Development of secured systems by mixing programs, specifications and proofs in an object-oriented programming environment. A case study within the FoCaLiZe environment

Damien Doligez, Mathieu Jaume, Renaud Rioboo

Development of secured systems by mixing programs, specifications and proofs in an object-oriented programming environment

A case study within the FoCaLiZe environment

Damien Doligez
Gallium
Inria, Rocquencourt, France
Damien.Doligez@inria.fr

Mathieu Jaume
SPI LIP6
Univ. P. & M. Curie, Paris, France
Mathieu.Jaume@lip6.fr

Renaud Rioboo
CPR CEDRIC
ENSIIE, Evry, France
Renaud.Rioboo@ensiie.fr

Abstract

FoCaLiZe is an object-oriented programming environment that combines specifications, programs and proofs in the same language. This paper describes how its features can be used to formally express specifications and to develop by stepwise refinement the design and implementation of secured systems, while proving that the implementation meets its specification or design requirements. We thus obtain a modular implementation of a generic framework for the definition of security policies together with certified enforcement mechanism for these policies.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks, Inheritance; D.2.4 [*Software/Program Verification*]: Correctness proofs, Formal methods; D.2.6 [*Programming Environments*]

General Terms Security, Languages, Verification

Keywords FoCaLiZe, security policies, enforcement mechanisms

1. Introduction

There is now a large collection of literature on security policies and secured systems: many policies and operational mechanisms have been proposed to ensure security in a system, each of them describing in a more or less formal way, within a particular specification language, a notion of information system suitable in a particular context together with some security requirements. Because of the variety of formalisms, it is rather difficult to reuse, refine, compare and compose these developments, for example when defining a variant of an existing development. To ease these tasks, it seems desirable to express such developments in a common framework and within a language that enables the development of modular programs and provides refinement mechanisms that allow working at different levels of abstraction in the same framework. Furthermore, the separation between the security policies and the operational mechanisms that are used to enforce them is now considered a main security design requirement: policies have to be specified independently of such mechanisms. In [19], a formal framework was introduced to deal with some of these issues; it is briefly described in section 3. The aim of the work presented in this paper is to implement a large part of this framework within a programming environment in order to obtain a development that allows defining security policies together with the operational mechanism that enforce them over transition systems and a formal certification of these mechanisms.

Nowadays, critical systems are evaluated according to some standards like the Common Criteria [12] or according to standards dedicated to particular domains (like the FIPS-140-3 specifying security requirements for cryptographic modules [16]). These standards often require the use of formal methods in order to ensure some safety and security properties needed of the systems in question. However, developing and evaluating such critical systems is a difficult task that requires advanced technical knowledge and large amounts of time. To make the task easier, we use the FoCaLiZe [17] integrated developement environment (IDE), which was conceived from the beginning to help build systems with high safety and security assurances, and which eases (and partially automates) the application of formal methods during the development cycle.

FoCaLiZe [17] provides an object-oriented functional language that allows writing specifications, programs, and

PLAS'12 June 15, Beijing, China.

the formal proofs that the programs meet their specifications. The object-oriented features of this language enable the development of an implementation by iterative refinement of its specification. Moreover, FoCaLiZe provides several automatic tools to ease the generation of programs from specifications [15], the generation of proofs [5], the generation of documentation, and the production of test suites [9, 10]. Using these tools together with an adequate methodology of development (as the one introduced in [3]) also makes the developments easier to formally evaluate according to the aforementioned standards. In the domains of safety and security, two main developments have been already done within FoCaLiZe: a full formalization of airport security regulations [14], and the implementation of a generic voter [2], which is a central equipment of all fault tolerant architectures, widely used for safety related systems.

2. FoCaLiZe

FoCaLiZe [17] is a programming environment that includes a language based on firm theoretical results [23], with a clear semantics and provides an efficient implementation – *via* translation to OCaml [22]. It has functional and object-oriented features and provides means for the programmers to write formal proofs of their code in a more or less detailed way within a declarative proof language based on the Zenon automatic theorem prover [5]. Zenon eases the task of writing formal proofs and translates them into Coq [13] for high-assurance checking. FoCaLiZe also provides powerful features (such as inheritance, parameterization and late-binding) that enable a stepwise refinement methodology to go from specification all the way down to executable code. Thus, FoCaLiZe unifies within the same language the formal modeling work, the development of the code, and the certification proofs.

2.1 Species

In FoCaLiZe, the primitive entity of a development is the *species*. Species are the nodes of the hierarchy of structures that makes up a development. A species can be seen as a set of methods grouping "things" related to the same concept. As in most modular design systems (i.e. object-oriented, abstract data types, etc.) the idea is to group a data structure with the operations on the data structure, the specification of these operations (in the form of properties), the representation requirements, and the proofs of the properties. Each method is identified by its name and can be either declared (primitive constants, operations and properties) or defined (implementation of operations, definition of properties and proofs of theorem). Moreover, we can distinguish three kinds of "methods": the carrier type, the programming methods and the logical methods (all the fields of our objects are called methods, be they types, data or code).

Carrier type The carrier, or representation type, is the concrete representation of the elements of the set underlying the structure defined by the species. The carrier is represented by the keyword `Self` inside the species and outside, by the name of the species itself, so that we identify the set with the structure, as usual in mathematics. Each species must have one unique carrier, but like all the other methods, it can be either declared or defined. A declared carrier is simply an abstract data type, while a defined one is a binding to a concrete type.

Programming methods These methods represent the constants and the operators of the structure. Declared methods are introduced by the keyword `signature`, defined methods are introduced by `let` and recursive definitions must be explicitely flagged with the keyword `rec`. The language used for the definitions is similar to the functional core of OCaml [22] (let-binding, pattern matching, conditional, higher order functions, etc), with the addition of a construction to call a method from a given structure. More precisely, the main syntactic constructions of the language are the following:

- abstraction with respect to a variable: `fun x -> ...`
- application of a function: `f(x)`
- call of a method `m` from a structure `c`: `c!m`
- call of a method `m` of the structure we are currently building: `Self!m` or just `m`

Logical methods These methods represent the properties of programming methods. In this context, the declaration of a logical method is simply the statement of a property, while the definition is a proof of this statement. In the first case, we speak of properties (`property`) that are still to be proved later in the development, while in the second case we speak of theorems (`theorem`). The language also allows logical definitions (`logical let`) to bind names to logical statements. The language used for the statements is composed of the basic logical connectors `and`, `or`, `->`, `<->`, `not`, and universal (`all`) and existential (`ex`) quantification over a FoCaLiZe type. Proofs in FoCaLiZe are written in a declarative format inspired by Lamport's work [11, 21]. A proof is a tree where the programmer introduces names (`assume`) and hypotheses (`hypothesis`), gives a statement to prove (`prove`) and then provides justification for the statement. This justification can be: (1) a "`conclude`" clause for fully automatic proof; (2) a "`by`" clause with a list of definitions, properties, hypotheses, previous theorems, and previous steps (subject to some scoping conditions) for use by the automatic prover; (3) a sequence of proofs (with their own assumptions, statements, and proofs) whose statements will be used by the automatic prover to prove the current statement. Hence, each step of a proof is independent of the others and can be reused in a similar context (this eases the maintenance of proofs and allows, for example, using exactly the same proof for a statement based on an hypothesis A and for the same statement

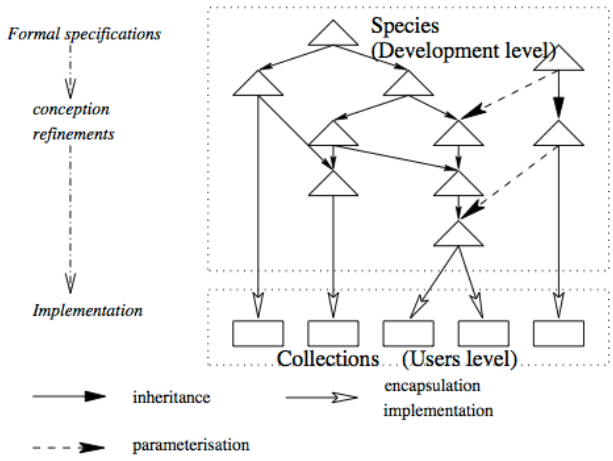

Figure 1. FoCaLiZe

based on a stronger hypothesis B, provided the automatic prover can make the inference from B to A).

2.2 Combining species

The main (object-oriented) features of FoCaLiZe are illustrated in Figure 1: a FoCaLiZe development is organized as a hierarchy which may have several roots. The upper levels of the hierarchy are built during the specification stage while the lower ones correspond to implementations.

Inheritance Using inheritance in FoCaLiZe, one can enrich a species with additional operations (methods) and redefine some methods of the parent species, but one can also get closer to a runnable implementation by providing explicit definitions to methods that were only declared in the parent. Note that the inheritance framework requires to perform static analysis in order to check coherence properties (inheritance lookup, resolution of multiple-inheritance conflicts, dependency analysis, type-checking, etc). In FoCaLiZe, classical object-oriented features have been restricted in order to avoid unsound constructions that can lead to inconsistencies when used carelessly. A species can inherit the declarations and definitions of one or several already defined species and is free to define or redefine any inherited method as long as such (re)definition does not change the type of the method.

Collections A collection is built upon a completely defined species. This means that every method must be defined. In other words, in a collection, every operation has an implementation, and every theorem is formally proved. In addition, a collection is "frozen": it cannot be used as a parent of a species in the inheritance graph. Moreover, to ensure modularity and abstraction, the carrier of a collection is hidden: seen from the outside, it becomes an abstract type. This means that any software component dealing with a collection will only be able to manipulate it through the operations it provides (i.e. its methods). This point is especially important since it prevents other software components from breaking representation invariants required by the internals of the collection.

Parameterization Besides inheritance, another important feature of FoCaLiZe is the ability to parameterize a species by collections parameters (whose "types" are given species. This mechanism allows using a species, not to embed its methods, but rather to use it as an "ingredient" to build a new structure by calling its methods explicitly.

2.3 Programming with FoCaLiZe

The computational part of FoCaLiZe is validated by its computer algebra library, mostly developed by R. Rioboo [6, 25] which implements mathematical structures up to multivariate polynomial rings and includes complex algorithms with performance comparable to the best computer algebra systems in existence. Furthermore, as we will see, this library is very useful when formalizing some security models based on partial orders, lattices or boolean algebras. Of course, nowadays, proof assistants provide some features for structuring code (module systems, type classes, etc), but most of them still cannot be used to obtain efficient programs. Compilation of FoCaLiZe developments leads to efficient OCaml programs (which are not obtained by extracting computational contents of proofs). It is this focus on efficiency that makes FoCaLiZe a real programming language. Hence, the main originality of FoCaLiZe is to provide an object-oriented programming language that allows mixing specifications, programs and proofs. To our knowledge, only the Agda [8] programming language, based on dependent types and compiling *via* Haskell, has a comparable mix of features. Note that the FoCaLiZe language is also based on a dependent type language, but with some restrictions on dependencies: for instance, a function cannot depend on a proof. By allowing such dependencies, we might get a better treatment of partial functions, but function redefinition would get trickier to handle because of logical clashes. In practice, this seems too difficult and we have rejected this possibility.

3. Security policies and secured systems

Several points of view exist on security policies, among which two main approaches can be distinguished: the rule-based approach (which consists in specifying the conditions under which an action is granted) and the property-based approach (which consists in specifying the security properties the policy aims to enforce). These two approaches are formally specified and compared in [19], and an operational mechanism for enforcing such policies over transition systems is defined together with the proof of its soundness. In this framework, one can characterize the various entities involved in the definition of a security policy together with their roles, thus providing a semantic specification of security policies. Several developments on access control poli-

cies and flow policies have been done within this framework: an operational mechanism that detects illegal information flows according to the flow policy induced by an access control policy is defined in [20], and the rule-based approach is considered in [7] by using rewrite systems. In this paper, we focus on the property-based approach, which provides for a clear distinction between a policy and its enforcement mechanism.

3.1 Security policies

Defining a security policy by following the property-based approach consists in characterizing secure elements of a set according to some security information. Thus, specifying a policy $\mathbb{P}$ first consists in defining a set $\mathbb{T}$ of "things" that the policy aims at controlling, called the security targets. These "things" can be the actions simultaneously done in the system, or some information about the entities of the system. Then a set $\mathcal{C}$ of security configurations is introduced: configurations correspond to the information needed to characterize secure elements of $\mathbb{T}$ according to the policy. Finally, the policy is specified by a binary relation $\Vdash$ between targets and configurations: $c \Vdash t$ means that the target t is secure according to the configuration c. Hence, a security policy is a triple:

$$\mathbb{P} = (\mathbb{T}, \mathcal{C}, \Vdash)$$

where $\mathbb{T}$ is a set of security targets, $\mathcal{C}$ is a set of security configurations and $\Vdash \subseteq \mathcal{C} \times \mathbb{T}$ is a relation specifying secure targets according to configurations.

For example, when dealing with access control policies, targets are sets of accesses simultaneously done in the system and we can represent accesses as triples (s, o, a) expressing that a subject $s \in \mathcal{S}$ has an access over an object $o \in \mathcal{O}$ according to an access mode $a \in \mathcal{A}$. Hence, in this context, the set $\mathbb{T}_A$ of targets is the powerset of the cartesian product $\mathcal{S} \times \mathcal{O} \times \mathcal{A}$. In this paper, we illustrate our developement with two classical access control policies.

The HRU policy [18] is a discretionary access control policy whose configurations are sets of authorized accesses, thus targets and configurations both specify sets of accesses and we have $\mathbb{T}_A = \mathcal{C}_{HRU}$. Hence, secure targets are defined as sets of accesses which are granted:

$$c \Vdash_{\mathbb{P}_{HRU}} t \Leftrightarrow t \subseteq c$$

We also consider the mandatory part of the Bell & La-Padula policy [4], whose configurations are tuples:

$$(\mathcal{L}, \preceq, f_o, f_s) \in \mathcal{C}_{BLP}$$

where $(\mathcal{L}, \preceq)$ is a partially ordered set of security levels (or sensitivities), and where $f_o : \mathcal{O} \to \mathcal{L}$ (resp. $f_s : \mathcal{S} \to \mathcal{L}$) associates a security level with each object (resp. each subject). Then, secure targets are sets of accesses such that the following two properties hold.

- (MAC property) Each read access over an object o is done by a subject whose security level is greater than or equal to $f_o(o)$ (in order to ensure a confidentiality policy).

$$\forall (s, o, \text{read}) \in t, f_o(o) \preceq f_s(s)$$

- (MAC$\star$ property) Each write access over an object o_1 is done by a subject whose read accesses are only done over objects o_2 such that $f_o(o_2) \preceq f_o(o_1)$ (in order to avoid information flows from high levels to low levels):

$$\forall (s, o_1, \text{write}) \in t, \forall (s, o_2, \text{read}) \in t, f_o(o_2) \preceq f_o(o_1)$$

3.2 Enforcement mechanism of security policies

Property-based policies can be used to ensure some security properties over reachable states of labelled transition systems (LTSs). We now define such an enforcement mechanism by showing how to "apply" a policy $\mathbb{P} = (\mathbb{T}, \mathcal{C}, \Vdash)$ "over" a LTS $\mathbb{S} = (\Sigma, \Sigma^0, L, \delta)$ (where Σ is the set of states, $\Sigma^0 \subseteq \Sigma$ is the set of initial states, L is the set of labels (or actions) and $\delta \subseteq \Sigma \times L \times \Sigma$ is the transition relation). Features of the system we want to obtain are described by $\mathbb{S}$ while security requirements of this system are specified by $\mathbb{P}$.

First, in order to define a secured system from $\mathbb{S}$ and $\mathbb{P}$, we have to define an interface between the system and the policy. This can be done by considering an interpretation:

$$I : \Sigma \to \mathbb{T}$$

mapping states of the system to targets of the policy. Then, according to this interpretation, a secured system can be defined by:

$$\mathbb{S}_{\mathbb{P}} = (\Sigma_{\mathbb{P}}, \Sigma^0_{\mathbb{P}}, L, \delta_{\mathbb{P}})$$

where

- $\Sigma_{\mathbb{P}} = \Sigma \times \mathcal{C}$ (configurations are used to monitor the system and a state of the secured system corresponds to a state of the initial system and a configuration of the policy),
- $\Sigma^0_{\mathbb{P}}$ is the set of secure initial states:

$$\Sigma^0_{\mathbb{P}} = \{(\sigma, c) \mid \sigma \in \Sigma^0 \wedge c \Vdash I(\sigma)\}$$

- $\delta_{\mathbb{P}}$ is the transition relation obtained from δ by removing transitions from secure states to non-secure states:

$$\delta_{\mathbb{P}} = \left\{ \begin{array}{l} (\sigma_1, c) \xrightarrow{l}_{\delta_{\mathbb{P}}} (\sigma_2, c) \mid \\ \sigma_1 \xrightarrow{l}_{\delta} \sigma_2 \wedge c \Vdash I(\sigma_1) \Rightarrow c \Vdash I(\sigma_2) \end{array} \right\}$$

Of course, it is easy to prove (by induction) that every reachable state (σ, c) of $\mathbb{S}_{\mathbb{P}}$ is secure according to $\mathbb{P}$ (i.e. is such that $c \Vdash I(\sigma)$). Hence, by following such an approach, the initial system is designed to ensure some features (for example, an access system allows opening and to closing accesses over objects) without taking into account security, while the

policy is intended to specify the desired security properties, which generally depend on external information about the configurations (for example, a policy specifying secure sets of opened accesses according to autorized accesses). Therefore, the definition of $\mathbb{S}_{\mathbb{P}}$ provides a generic method to automatically enforce an abstract security policy on a system.

4. Development of secured systems within FoCaLiZe

This section describes the implementation within FoCaLiZe of the framework presented in section 3. More precisely, we show here how to take advantage of the features provided by FoCaLiZe to easily produce a modular, certified and efficient implementation of a secured system. As we said, none of these features are new, but it is their combination within the same programming language that is original. Throughout this section, only the most significant parts of the code will be shown (technical details will be replaced with "...").

4.1 Transition systems

We first briefly describe the implementation of LTSs from which secured systems will be defined. In Table 1, we introduce the species `Trans_syst` that describes a transition system parameterized by a set `S` of states (species `State` containing the declaration of a predicate characterizing initial states), a set `L` of labels (species `Label`), and an abstract notion `SLS` of triples (species `Abstract_triple` providing generic methods over triples). These species inherit from the species `Setoid` (non-empty set with an equivalence relation).

```
species State =
 inherit Setoid;
 signature is_initial : Self -> bool;
end;;
species Label = inherit Setoid; end;;
```

The parameter `SLS` can be viewed as a simple data structure, provided to methods of the species `Trans_syst` to manipulate triples. This is needed when defining sequences of transitions of a LTS (which are lists of triples, each triple (σ_1, l, σ_2) denoting a transition, labeled l, from state σ_1 to state σ_2). Since a transition system can be completely defined by the parameters and the methods introduced by the species `Trans_syst`, we use `unit` as its carrier type. Note that this species describes a single transition system, not a collection of transition systems. The (declared) method `delta` corresponds to the transition relation of the system; it is used to define the logical methods that specify determinism and completeness of this transition transition relation. The logical method `is_reachable` specifies the set of reachable states of this transition system: a state x is reachable iff there exists a sequence of transitions starting from an initial state and ending by x. This method is based on the recursive boolean function `is_path_from_init_to` that characterizes sequences of transitions starting from an initial state and leading to a given state:

- `is_path_from_init_to([],`x`)=true` means that the empty sequence of transition leads to the state x (i.e. x is an initial state).
- `is_path_from_init_to(`h`::`ℓ`,`x`)` $=$ `true` means that h is a triple $(y, l, x) \in \delta$ and ℓ is a sequence of transitions starting from an initial state and leading to the state y.

These two properties are introduced as theorems (automatically proved from the definition of `is_path_from_init_to`) and are our specification of the `is_path_from_init_to` method. Of course, introducing theorems corresponding to these properties (which are very similar to the definition) may seem redundant. However, if the `is_path_from_init_to` method is redefined through inheritance, proofs of these theorems will be erased by the FoCaLiZe compiler and will have to be re-done with the new definition. Therefore, these theorems can be used during the proofs of other results, which avoids depending on the definition of `is_path_from_init_to` and thus allows redefining it without invalidating these proofs (only the proofs of the specification of `is_path_from_init_to` have to be done again). Remark that the inheritance mechanisms of object-oriented programming are not harmless when dealing with theorems: the main point here is that the redefinition of a function during inheritance may invalidate some proofs which are then erased and must be redone in the new context. In [24], such issues are discussed and a methodology is proposed to minimize the impact of redefinitions on proofs.

In practice, transition systems are often defined by a transition function $\tau : \Sigma \times L \to \Sigma$. Hence, we also introduce the species `Op_trans_syst` that inherits from `Trans_syst` and adds the declaration of a transition function from which the declared method `delta` can now be defined:

$$(\sigma_1, l, \sigma_2) \in \delta \Leftrightarrow \tau(\sigma_1, l) = \sigma_2$$

The definition of `Op_trans_syst` also gives the properties (`deterministic` and `complete`) that this definition leads to a deterministic and complete transition relation, along with the proofs of these properties.

Example: access system Based on the specification of LTSs within FoCaLiZe, we now build an implementation of access systems. An access system provides a way for users of a system (the subjects in $\mathcal{S}$) to access sets of services or resources (the objects in $\mathcal{O}$) according to some access modes (in $\mathcal{A}$). Subjects, objects and access modes are specified by finite sets:

```
species Subject = inherit Finite_set; end;;
species Object = inherit Finite_set; end;;
species Access_mode = inherit Finite_set; end;;
```

and states are represented by finite parts of the cartesian product $\mathcal{S} \times \mathcal{O} \times \mathcal{A}$. Hence, the species `State_ac` is parameterized by the subjects, the objects and the access modes, but also by the implementations of triples and finite subsets:

```
species Trans_syst (S is State, L is Label, SLS is Abstract_triple(S, L, S)) =
  inherit Basic_object;
  representation = unit;
  signature delta : S -> L -> S -> bool;
  logical let is_deterministic =
   all x y z : S, all l : L, ( delta (x, l, y) /\ delta (x, l, z) ) -> S!equal (y, z);
  logical let is_complete = all x : S, all l : L, ex y : S, delta (x, l, y);
  let rec is_path_from_init_to(p, x) = match p with
   | [] -> S!is_initial (x)
   | h :: q -> S!equal (SLS!third(h), x) && delta (SLS!first(h), SLS!second(h), x)
               && is_path_from_init_to (q, SLS!first(h));;
  theorem is_path_from_init_to_base: all x: S, is_path_from_init_to([], x) <-> S!is_initial(x)
  proof = by definition of is_path_from_init_to;
  theorem is_path_from_init_to_ind: all x: S, all h: SLS, all q: list(SLS),
       is_path_from_init_to((h :: q), x)
   <-> S!equal (SLS!third(h), x) /\ delta (SLS!first(h), SLS!second(h), x)
       /\ is_path_from_init_to (q, SLS!first(h))
  proof = by definition of is_path_from_init_to;
  logical let is_reachable (x) = ex p : list (SLS), is_path_from_init_to (p, x);
... end;;

species Op_trans_syst (S is State, L is Label, SLS is Abstract_triple(S, L, S)) =
  inherit Trans_syst (S, L, SLS);
  signature transition : S -> L -> S;
  let delta(s1, l, s2) = S!equal(transition(s1, l), s2);
  theorem deterministic : is_deterministic   proof = ...
  theorem complete : is_complete   proof = ...
... end;;
```

Table 1. Transition systems

```
species State_ac(
 S is Subject, O is Object, A is Access_mode,
 SOA is Abstract_triple(S, O, A),
 PSOA is Finite_parts(SOA)) ...
```

This species inherits from the species `State`, defines one initial state (the empty set) and specifies the carrier type as the representation of finite subsets of the set of triples (obtained from the parameter `PSOA`). Labels of access systems are just pairs $\langle +, (s, o, a)\rangle$ and $\langle -, (s, o, a)\rangle$ expressing that a subject s is going to to access $(+)$ or to release its access $(-)$ to the object o according to the mode a. This is specified by the species:

```
species Label_ac(
 S is Subject, O is Object, A is Access_mode,
 SOA is Abstract_triple(S, O, A)) ...
```

inheriting from `Label`. Now, from the following definition of the transition function:

$$\tau_{ac}(A, l) = \begin{cases} A \cup \{(s, o, a)\} & \text{if } l = \langle +, (s, o, a)\rangle \\ A \setminus \{(s, o, a)\} & \text{if } l = \langle -, (s, o, a)\rangle \end{cases}$$

obtained by using operations over finite subsets, it becomes possible to define a species:

```
species Op_trans_syst_ac(
 S is Subject,O is Object, A is Access_mode,
 SOA is Abstract_triple(S, O, A),
 L is Label_ac(S, O, A, SOA),
 PSOA is Finite_parts(SOA),
 St is State_ac(S, O, A, SOA, PSOA),
 SLS is Abstract_triple(St, L, St)) ...
```

that inherits from `Op_trans_syst(St, L, SLS)` and implements a complete and deterministic access system.

4.2 Secured systems

4.2.1 Secure states and Secured systems

Formalizing the framework introduced in section 3 within FoCaLiZe has led us to generalize it in order to obtain a generic and modular implementation. Indeed, the definition of a secured system is strongly related to the notion of secure state. We reuse here the notations of section 3, where the notion of secure state is defined by considering security policies and the states of a secured system are defined as the product $\Sigma \times \mathcal{C}$ (where $\mathcal{C}$ corresponds to the set of configurations of the policy), thus only secured systems obtained from a system and a policy can be described by following this approach. However, the construction of a secured system is independent of the way secure states are defined. Hence, we introduce here a more abstract notion of secure state that specifies what are states of secured systems (this abstract notion will be instantiated by $\Sigma \times \mathcal{C}$ when considering policies).

This allows using our development with other approaches that define a notion of secure state.

From an abstract point of view, the set Σ' of states of a secured system is obtained from the set Σ of states of the system we want to make secure and a predicate Ω over Σ that characterizes which stats are secure. In fact, the set Σ' can be viewed as a structure over Σ. Hence, we define the species `Sec_state` deriving Σ' from the parameter Σ. This species contains the declaration of the method `state` that maps back to the state $\sigma \in \Sigma$ from which a state $\sigma' \in \Sigma'$ was obtained: `state` corresponds to a projection function $I_S : \Sigma' \to \Sigma$. The converse method `make` is also declared to characterize the "decoration" that was added to a state $\sigma \in \Sigma$ to obtain a state $\sigma' \in \Sigma'$: if $\sigma' \in \Sigma'$ has been obtained from $\sigma_0 \in \Sigma$, then $\texttt{make}(\sigma', \sigma) = \sigma''$ means that the "decoration" which has been added when transforming σ_0 into σ' is the same as the one used to transform σ into σ''. Finally, we declare the method `is_secure` (corresponding to our Ω predicate) that characterizes the secure states of Σ'. It allows us to define the initial states of Σ' (method `is_initial` inherited from the species `State`) as the set:

$$\left\{\sigma' \in \Sigma' \mid I_S(\sigma') \in \Sigma^0 \wedge \Omega(\sigma')\right\}$$

Therefore, σ' is initial iff it is secure according to Ω and it has been obtained from an initial state $\sigma \in \Sigma^0$.

```
species Sec_state (S is State) =
 inherit State;
 signature state : Self -> S;
 signature make : Self -> S -> Self;
 signature is_secure : Self -> bool;
 property make_spec : all x: Self,
   equal(make(x, state(x)), x);
 let is_initial  (x : Self) =
   S!is_initial (state (x)) && is_secure (x);
... end;;
```

From this abstract notion of secure state, it becomes possible to implement secured systems (see table 2). First the species `Sec_trans_syst` is defined: it is parameterized by a transition system `Sys` (obtained from a set `St` of states and a set `L` of labels) and a notion of secure states `Ss` (obtained from `St`), and specifies (by inheritance) a transition system whose states are in `Ss` and whose labels are in `L`. The transition relation δ_Ω of this system is defined from the transition relation δ of `Sys` as follows:

$$\delta_\Omega = \left\{ \begin{array}{l} \sigma_1 \xrightarrow{l}_{\delta_\Omega} \sigma_2 \mid \\ I_S(\sigma_1) \xrightarrow{l}_{\delta} I_S(\sigma_2) \wedge \Omega(\sigma_1) \Rightarrow \Omega(\sigma_2) \end{array} \right\}$$

This definition allows us to prove the theorem asserting that each reachable state of this system is secure: the proof of the theorem `reachable_is_secure` is obtained by telling the Zenon automatic theorem prover to use the definition of reachable states (based on the method `is_path_from_init_to`), and the theorem `all_secure` (which asserts that if ℓ is a list representing a sequence of transitions that starts with an initial state and ends with a state x, then x is secure). As we can see in table 2, the proof of `all_secure` is obtained by induction over ℓ:

- `<1>1` is the proof of the base case obtained when ℓ is the empty list `[]` by telling Zenon to use the theorem `is_path_from_init_to_base` (which is part of the specification of `is_path_from_init_to` and entails that x is an initial state), and the property `Ss!is_initial_spec`, proved in the species `Sec_state`, which specifies that initial states are secure (here again, the proof does not depend on the definition of `is_initial` but only on its specification).
- `<1>2` is the proof of the inductive step obtained when ℓ is non-empty by telling Zenon to use the definition of δ_Ω (the method `delta` of the species `Sec_trans_syst`), the theorem `is_path_from_init_to_ind` (which is part of the specifcation of the method `is_path_from_init_to`, as described on page 5), the specification of projection operators and equality over triples, the reflexivity property of equality over labels, and a property on the equality defined over states (expressing that this relation is a congruence for the transition relation).

As we can see, proving a property within FoCaLiZe can usually be done just by identifying the definitions that must be unfolded and the properties from which Zenon can automatically build the proof.

Intuitively, δ_Ω is obtained by removing from δ all the "non-secure" transitions. Hence, even if δ is complete, δ_Ω is not necessarily complete and we have to refine this definition in order to obtain a complete and deterministic transition system. This can be done by replacing "non-secure" transitions from a state σ with the "identity" transition from σ (other approaches can be defined: for example, in some cases, it may be useful to introduce an element $\perp$ denoting an error state obtained when trying to perform a forbidden action in a system). Thus, we define the transition function $\tau_\Omega : \Sigma' \times L \to \Sigma'$ from the transition function $\tau : \Sigma \times L \to \Sigma$ (of the initial system) as follows:

$$\tau_\Omega(\sigma_1, l) = \left\{ \begin{array}{l} \texttt{make}(\sigma_1, \tau(I_S(\sigma_1), l)) \\ \quad \text{if } \Omega(\sigma_1) \Rightarrow \Omega(\texttt{make}(\sigma_1, \tau(I_S(\sigma_1), l))) \\ \sigma_1 \text{ otherwise} \end{array} \right.$$

By following such an approach, we define the species `Sec_op_trans_syst`, which implements a complete and deterministic transition system parameterized by the complete and deterministic system `Sys` that we want to make secure, and a notion `Ss` of secure states, and which implements (by inheritance) both a secured system and a deterministic and complete system. However, since the transition relation of the complete and deterministic system is defined from τ_Ω,

this transition relation corresponds now to the relation:

$$\begin{aligned}\delta'_\Omega = & \left\{\sigma_1 \xrightarrow{l}_{\delta'_\Omega} \sigma_2 \mid \sigma_1 \xrightarrow{l}_{\delta_\Omega} \sigma_2\right\} \\ \cup & \left\{\sigma_1 \xrightarrow{l}_{\delta'_\Omega} \sigma_1 \mid \nexists \sigma_2\ \sigma_1 \xrightarrow{l}_{\delta_\Omega} \sigma_2\right\}\end{aligned}$$

which does not coincide with the definition of δ_Ω. In fact, the method `delta` of the species `Sec_trans_syst` is redefined here. Therefore, the proof of `all_secure` (which depends on the definition of the transition relation) from the species `Sec_trans_syst` is erased by FoCaLiZe, and we have to give another proof to take the new definition into account.

4.2.2 Secured systems and Policies

We show here how to instantiate our generic notion of secured system by considering security policies as introduced in section 3.

Security policies We first introduce the species implementing a policy, parameterized by a setoid of targets and a setoid of configurations. It inherits from the species of binary relations. We declare the method `secure` to characterize secure targets according to configurations. It is used to define the `relation` method inherited from the binary relation species.

```
species Target = inherit Setoid; end;;
species Configuration = inherit Setoid; end;;
species P_policy
 (A is Target, C is Configuration)=
 inherit Relation (A, C);
 signature secure: A -> C -> bool;
 let relation(a, c) = secure(a, c);
end;;
```

Example: Access control policies As illustrated by figure 2, the species `P_policy` is the root of a hierarchy of security policies. For example, the species `P_policy_ac` implements access control policies and can be defined by simply specifying targets as an abstract ternary relation between subjects, objects and access modes:

```
species P_policy_ac (
 S is Subject, O is Object,A is Access_mode,
 R is Ternary_relations(S, O, A),
 T is Target_ac(S, O, A, R),
 C is Configuration)=
inherit P_policy (T, C);
end;;
```

Hence, at this level, nothing is assumed on the representation of sets of accesses and this species can be instantiated with any implementation of sets of accesses that inherits from `Ternary_relations(S, O, A)`. Then, the HRU policy (introduced in section 3) can be specified as follows:

```
species P_policy_hru (
 S is Subject, O is Object, A is Access_mode,
 R is Ternary_relations(S, O, A),
 T is Target_ac (S, O, A, R),
 C is Configuration_ac (S, O, A, R)) =
inherit P_policy_ac (S, O, A, R, T, C);
let secure (t, c) =
 R!is_contained(T!as_relation(t),C!as_relation(c));
... end;;
```

Configurations of this policy are also sets of (authorized) accesses represented by the same abstract notion of ternary relation than targets (specified by the parameter `R`), and from which the method `secure` can be defined: t is secure according to c iff t, viewed as a relation, is contained in c, also viewed as a relation (species `Target_ac` and `Configuration_ac` both contain the method `as_relation:Self -> R`).

We can also refine the definition of `P_policy_ac` by considering a particular set of access modes: for example $\mathcal{A} = \{\text{read}, \text{write}\}$.

```
species Access_mode_rw =
 inherit Access_mode;
 signature read: Self;
 signature write: Self;
end;;
species P_policy_ac_rw (
 S is Subject, O is Object, A is Access_mode_rw,
 R is Ternary_relations(S,O,A),
 T is Target_ac (S, O, A, R),
 C is Configuration)=
inherit P_policy_ac (S, O, A, R, T, C);
end;;
```

Now, we can refine again the definition of the HRU policy by using multiple inheritance as follows:

```
species P_policy_hru_rw(
 S is Subject,O is Object, A is Access_mode_rw,
 R is Ternary_relations(S,O,A),
 T is Target_ac (S, O, A, R),
 C is Configuration_ac (S, O, A, R)) =
inherit P_policy_ac_rw(S, O, A, R, T, C),
        P_policy_hru(S, O, A, R, T, C);
end;;
```

As we can see here, parameterization and inheritance are powerful mechanisms for building new components from existing components.

The species `P_policy_ac_rw` can also be refined by considering the Bell & LaPadula policy (introduced in section 3). Security levels are obtained by inheritance from the species `Partial_order` of the FoCaLiZe library, and are used to define the species of configurations:

```
species Security_level=inherit Partial_order; end;;
species Configuration_BLP
 (S is Subject, O is Object, L is Security_level)=
 inherit Configuration;
 signature fs: Self -> S -> L;
 signature fo: Self -> O -> L;
end;;
```

The Bell & LaPadula policy can now be implemented:

```
species Sec_trans_syst (St is State, L is Label, SLT is Abstract_triple(St, L, St),
    Sys is Trans_syst (St, L, SLT), Ss is Sec_state (St), SLS is Abstract_triple(Ss, L, Ss)) =
  inherit Trans_syst (Ss, L, SLS);
  let delta (x : Ss, l : L, y : Ss) =
   Sys!delta (Ss!state (x), l, Ss!state (y)) && ( ~~ Ss!is_secure (x) || Ss!is_secure (y) );
  theorem all_secure: all l: list(SLS), all x: Ss, is_path_from_init_to(l, x) -> Ss!is_secure (x)
  proof =
   <1>1 prove all x: Ss, is_path_from_init_to([], x) -> Ss!is_secure (x)
        by property is_path_from_init_to_base, Ss!is_initial_spec
   <1>2 prove all l: list(SLS),
          (all z: Ss, is_path_from_init_to(l, z) -> Ss!is_secure(z))
        -> all x: Ss, all t: SLS, is_path_from_init_to((t :: l), x) -> Ss!is_secure(x)
        by definition of delta
           property is_path_from_init_to_ind, L!equal_reflexive, SLS!is_first, SLS!is_second,
                    SLS!is_third, SLS!equal_spec, delta_equal_compat
   <1>3 conclude ;
  theorem reachable_is_secure: all x: Ss, is_reachable(x) -> Ss!is_secure(x)
  proof = by definition of is_reachable property all_secure;
... end;;

species Sec_op_trans_syst(St is State, L is Label, SLT is Abstract_triple(St, L, St),
    Sys is Op_trans_syst(St, L, SLT), Ss is Sec_state (St), SLS is Abstract_triple(Ss, L, Ss) ) =
  inherit Sec_trans_syst (St, L, SLT, Sys, Ss, SLS), Op_trans_syst (Ss, L, SLS);
  let transition(x, l) = let s = Sys!transition(Ss!state(x), l) in let y = Ss!make(x, s) in
   if ((~~ Ss!is_secure(x)) || Ss!is_secure(y)) then y else x;
  proof of all_secure = ...
... end;;
```

Table 2. Secured systems

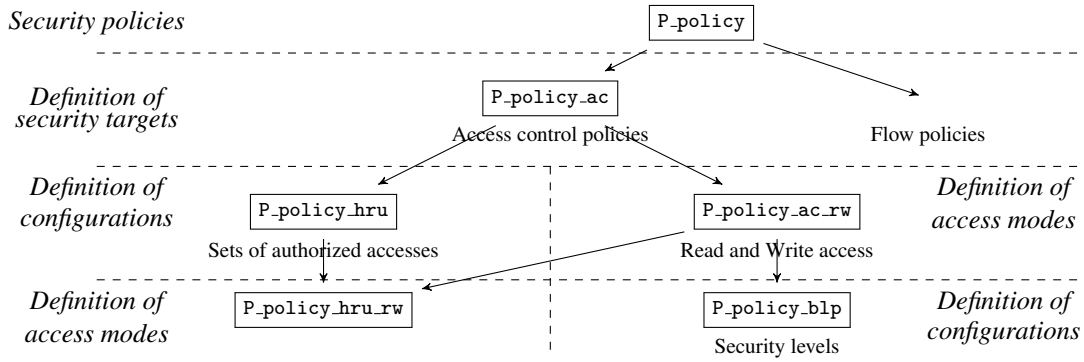

Figure 2. Hierarchy of development of security policies

```
species P_policy_BLP(
 S is Subject, O is Object,
 L is Security_level, A is Access_mode_rw,
 SOA is Abstract_triple(S, O, A),
 PSOA is Finite_parts(SOA),
 R is Relations_by_sets_of_triples(S,O,A,SOA,PSOA),
 T is Target_ac_by_sets_of_triples(S,O,A,SOA,PSOA,R),
 C is Configuration_BLP(S, O, L)) =
inherit P_policy_ac_rw(S, O, A, R, T, C);
representation = unit ;
logical let mac(c, r) = all s: S, all o: O,
    (R!relation(r, s, o, A!read)
     -> L!leq(C!fo(c, o), C!fs(c, s));
logical let mac_star(c, r) = ...
logical let is_secure(t, c) =
    mac(c, T!as_relation(t))
 /\ mac_star(c, T!as_relation(t));
let rec mac_secure(l, c) =  match l with
 | [] -> true
 | h::t ->
 (~~( A!equal(SOA!third(h), A!read)) ||
  L!leq(C!fo(c,SOA!second(h)),
        C!fs(c,SOA!first(h))))
 && mac_secure(t, c);
```

```
theorem mac_correct: all c: C, all t: T,
      mac_secure(PSOA!to_list(T!as_part(t)),c)
 <->  mac(c, T!as_relation(t))
 proof = ...
let rec mac_star_secure(l, c) = ...
theorem mac_star_correct: ...
let secure(t,c)=
 let l = PSOA!to_list(T!as_part(t))
 in mac_secure(l, c) && mac_star_secure(l, c)
theorem secure_is_correct: all t: T, all c: C,
    secure(t, c) <-> is_secure(t, c)
  proof = ...
end;;
```

Note that while the HRU policy can be specified with a very abstract notion of set of accesses (only an inclusion test is needed), the BLP policy needs a more concrete specification of sets of accesses because we have to be able to consider each access and to use projection functions to specify properties on subjects, objects and access modes occurring in accesses. This is obtained by taking as parameters ternary relations represented as finite sets of triples (species `Relations_by_sets_of_triples`). This makes a difference when comparing policies "on paper" and implementations of these policies within FoCaLiZe. The species `P_policy_BLP` defines two predicates (`mac` and `mac_star`) that correspond exactly to the properties introduced in section 3; they characterize secure targets according to configurations: they are expressed by considering targets as ternary relations. However, the method `secure` declared in the species `P_policy` (from which `P_policy_BLP` inherits) is a boolean function and needs an operational definition. Hence, we define the methods `mac_secure` and `mac_star_secure` which are functions, where targets are considered as finite sets represented by lists. Of course, we prove the equivalence between each predicate and its corresponding function.

Secure states and security policies We show here how to refine the species of states of a secured system by considering a security policy. As we said in section 3, states of the secured system are pairs belonging to the cartesian product $\Sigma_{\mathbb{P}} = \Sigma \times \mathcal{C}$ and we introduce the species `Sec_state_pol` (specifying a set Σ' of states) parameterized by a policy $\mathbb{P}$ and the set of states Σ of the system we want to make secure. It inherits from the species of cartesian products (instantiated with Σ and $\mathcal{C}$ and from the species `Sec_state(`Σ`)`. In this context the method `state` can be defined by the function $I_S : \Sigma_{\mathbb{P}} \rightarrow \Sigma$ such that:

$$\forall (\sigma, c) \in \Sigma_{\mathbb{P}}, I_S((\sigma, c)) = \sigma$$

Conversely, the method `make` is defined by:

$$\texttt{make}((\sigma_1, c), \sigma_2) = (\sigma_2, c)$$

Finally, secure states of Σ' are characterized by introducing an interpretation $I : \Sigma \rightarrow \mathbb{T}$ as explained in section 3. This leads us to define the predicate Ω by:

$$\Omega((\sigma, c)) \Leftrightarrow c \Vdash I(\sigma)$$

```
species Sec_state_pol(
 A is Target, C is Configuration,
 P is P_policy (A, C),
 S is State) =
inherit Cartesian_product(S, C), Sec_state(S);
let state(s) = first(s);
let configuration(s) = second(s);
let make(s_s, s_i) = pair(s_i, configuration(s_s));
proof of make_spec =  ...
signature interpretation : S -> A;
let is_secure (x : Self) =
 P!secure(interpretation(state(x)),
          configuration(x));
... end;;
```

Enforcement mechanism of security policies We are now ready to implement secured systems by considering policies. Here again, thanks to the parameterization and inheritance mechanisms, this can be done just by combining already defined species. Indeed, the species implementing a secured system that corresponds to the system $\mathbb{S}_{\mathbb{P}}$ defined in section 3 can directly be obtained by considering secure states built from a policy as follows:

```
species Sec_trans_syst_pol(
 A is Target,
 C is Configuration,
 P is P_policy(A, C),
 St is State,
 L is Label,
 SLS is Abstract_triple(St, L, St),
 Sys is Trans_syst(St, L, SLS),
 Ss is Sec_state_pol(A, C, P, St),
 SsLS is Abstract_triple(Ss, L, Ss)) =
inherit Sec_trans_syst(St, L, SLS, Sys, Ss, SsLS);
end;;
```

Similarly, the deterministic and complete system secured by a policy is obtained as follows:

```
species Sec_op_trans_system_pol(
 A is Target,
 C is Configuration,
 P is P_policy(A, C),
 St is State,
 L is Label,
 SLS is Abstract_triple(St, L, St),
 Sys is Op_trans_syst(St, L, SLS),
 Ss is Sec_state_pol(A, C, P, St),
 SsLS is Abstract_triple(Ss, L, Ss)) =
inherit
 Sec_trans_syst_pol(A,C,P,St,L,SLS,Sys,Ss,SsLS),
 Sec_op_trans_syst(St,L,SLS,Sys,Ss,SsLS);
end;;
```

Hence, the transition relation of such a system is defined by:

$$\delta'_{\mathbb{P}} = \left\{ \begin{array}{l} (\sigma_1, c) \xrightarrow{l}_{\delta'_{\mathbb{P}}} (\sigma_2, c) \mid \\ \sigma_1 \xrightarrow{l}_{\delta} \sigma_2 \wedge c \Vdash I(\sigma_1) \Rightarrow c \Vdash I(\sigma_2) \end{array} \right\} \cup \left\{ \begin{array}{l} (\sigma_1, c) \xrightarrow{l}_{\delta'_{\mathbb{P}}} (\sigma_1, c) \mid \\ \sigma_1 \xrightarrow{l}_{\delta} \sigma_2 \wedge c \Vdash I(\sigma_1) \wedge \neg c \Vdash I(\sigma_2) \end{array} \right\}$$

and, as we said, does not exactly correspond to $\delta_{\mathbb{P}}$.

Example: Secured access system We can now obtain a secured access system from the species introduced above. For example, we can build the following deterministic and complete system from the species `Op_trans_syst_ac` (specifying what is an access system) and `P_policy_BLP` (specifying the Bell & LaPadula policy) as follows:

```
species Sec_syst_blp(
 S is Subject,
 O is Object,
 Le is Security_level,
 A is Access_mode_rw,
 SOA is Triples_by_pairs(S, O, A),
 PSOA is Finite_parts(SOA),
 St is State_ac(S, O, A, SOA, PSOA),
 La is Label_ac(S, O, A, SOA),
 SLS is Abstract_triple(St, La, St),
 R is Relations_by_sets_of_triples(S,O,A,SOA,PSOA),
 T is Target_ac_by_sets_of_triples(S,O,A,SOA,PSOA,R),
 C is Configuration_BLP(S, O, Le),
 P is P_policy_BLP(S, O, Le, A, SOA, PSOA, R, T, C),
 Sys is Op_trans_syst_ac(S,O,A,SOA,La,PSOA,St,SLS),
 Ss is Sec_state_pol(T, C, P, St),
 SsLS is Abstract_triple(Ss, La, Ss)) =
inherit Sec_op_trans_syst_pol
        (T, C, P, St, La, SLS, Sys, Ss, SsLS);
end;;
```

Here again, parameters both specify the "ingredients" needed to define the species and the way these "ingredients" are combined at several levels of abstraction (for example, triples built from subjects, objects and access modes are here represented by pairs of the form $(s, (o, a))$ which is a very concrete representation, while triples built from states, labels and states are just specified at a higher level of abstraction.

5. Conclusion

Many developments have been done on security policies and enforcement mechanisms but few of them are formalized within a generic framework that allows reusing, comparing and composing such developments. Furthermore, operational mechanims used to enforce policies are usually not designed independently of the policy they are intended to enforce, which does not lead to a clear separation between the specification of the desired security properties and the mechanisms (i.e. the method used to implement the policy) that enforce these properties. In [26], by using the B refinement process [1], an enforcement mechanism of a policy in a TCP/IP network is obtained from an abstract specification of this policy. Our approach is similar: in this paper, we have both introduced an abstract framework for specifying security policies and a sound generic operational mechanism to enforce such policies over transition systems.

In order to formally certify our enforcement mechanism, our implementation is done within the FoCaLiZe programming environment, and yields modular programs. Indeed, thanks to its object-oriented features, FoCaLiZe allows us to describe the same notion at several levels of abstraction (at each level, we focus on the main operations we want to specify or to define). Moreover, many software components implemented within FoCaLiZe (i.e. species or collections) can be directly built by inheritance and parameterization from already defined components. Hence, such developments can easily be reused in different contexts. Last but not least, it should be noted that the proofs were easy to do, thanks to the Zenon automatic theorem prover.

Of course, all these features facilitate the development of certified programs and therefore FoCaLiZe is particularly well-suited to develop libraries for secure applications. In fact, the claim behind FoCaLiZe is that formal developments increase confidence in the final code. One of the main characteristics of critical software is that it is subject to the approval of a safety/security authority before its commissioning. These authorities have defined requirements explaining what should be an acceptable software and its related life cycle process for their own domain. For this reason, getting a high confidence in produced code, and making it possible for the safety/security authority to acquire this confidence is an important task, for which FoCaLiZe brings solutions. Very important is the possibility in FoCaLiZe to have specification, implementation and proofs *within the same language*, since it eliminates the errors introduced between layers, at each switch between languages, during the development cycle. Other frameworks like Atelier B [1] also aims to implement tools for making formal development a reality. FoCaLiZe doesn't follow the same path, trying to keep the means of expression close to what engineers usually know: a programming language. Moreover, instead of having its own system for proofs validation, FoCaLiZe makes use of external tools, leaving the task of handling proof automation and verification outside its scope and reaping the benefits of research performed by others in these specific domains.

As future work, we plan to implement some comparison mechanisms between security policies in order to formalize within FoCaLiZe the flow-based interpretation of access control policies, as defined in [20]. Indeed, we are currently formalizing software requirements expressed in the standard FIPS 140-3, which specifies security requirements of a cryptographic module. This leads us to structure these requirements and to express them as security policies. Our aim is

to formally prove some abstract confinement, confidentiality and integrity properties induced by these concrete requirements.

Acknowledgments

Many thanks to Thérèse Hardin and François Pessaux for enlightening discussions about this work.

References

[1] J. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] P. Ayrault, T. Hardin, and F. Pessaux. Development of a generic voter under Focal. In *Tests and Proofs, Third Int. Conf., TAP 2009, Proceedings*, volume 5668 of *LNCS*, pages 10–26. Springer, 2009.

[3] P. Ayrault, T. Hardin, and F. Pessaux. Development life-cycle of critical software under focal. *Electr. Notes Theoretical Computer Science*, 243:15–31, 2009.

[4] D. Bell and L. LaPadula. Secure Computer Systems: a Mathematical Model. Technical Report MTR-2547 (Vol. II), MITRE Corp., Bedford, MA, 1973.

[5] R. Bonichon, D. Delahaye, and D. Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. In *Logic for Programming, Artificial Intelligence, and Reasoning, 14th Int. Conf., LPAR*, volume 4790 of *LNCS*, pages 151–165. Springer, 2007.

[6] S. Boulmé, T. Hardin, and R. Rioboo. Some hints for polynomials in the Foc project. In *9th Symp. on the Integration of Symbolic Computation and Mechanized Reasoning, Calculemus 2001*, 2001.

[7] T. Bourdier, H. Cirstea, M. Jaume, and H. Kirchner. Formal specification and validation of security policies. In *Foundations & Practice of Security, FPS 2011*, volume 6888 of *LNCS*, pages 148–163. Springer, 2011.

[8] A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda - A functional language with dependent types. In *Theorem Proving in Higher Order Logics, 22nd Int. Conf., TPHOLs 2009, Proceedings*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009.

[9] M. Carlier and C. Dubois. Functional testing in the Focal environment. In *Tests and Proofs, Second Int. Conf., TAP 2008, Proceedings*, volume 4966 of *LNCS*, pages 84–98. Springer, 2008.

[10] M. Carlier, C. Dubois, and A. Gotlieb. Constraint reasoning in FocalTest. In *ICSOFT 2010 - Proceedings of the Fifth Int. Conf. on Software and Data Technologies, Volume 2*, pages 82–91. SciTePress, 2010.

[11] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. A TLA$^+$ proof system. In G. Sutcliffe, P. Rudnicki, R. Schmidt, B. Konev, and S. Schulz, editors, *Proc. of the LPAR Workshop Knowledge Exchange: Automated Provers and Proof Assistants (KEAPPA'08)*, number 418 in CEUR Workshop Proceedings, pages 17–37, 2008.

[12] Common Criteria. *Common Criteria for Information Technology Security Evaluation, Norme ISO 15408 – Version 3.0 Rev 2*, 2005.

[13] Coq. *The Coq Proof Assistant, Tutorial and reference manual*. INRIA – LIP – LRI – LIX – PPS, 2010. Distribution available at: `http://coq.inria.fr/`.

[14] D. Delahaye, J. Étienne, and V. Donzeau-Gouge. Certifying airport security regulations using the Focal environment. In *FM 2006: 14th Int. Symp. on Formal Methods*, volume 4085 of *LNCS*, pages 48–63. Springer, 2006.

[15] D. Delahaye, C. Dubois, and P. Tollite. Génération de code fonctionnel certifié à partir de spécifications inductives dans l'environnement Focalize. In *21th Journées Francophones des Langages Applicatifs*, 2010.

[16] FIPS. *Security Requirements for Cryptographic Modules, FIPS 140-3*. National Institute for Standards and Technology, 2009.

[17] Focalize. *Focalize, Tutorial and reference manual*. LIP6 – INRIA – CEDRIC, 2010. Distribution available at: `http://focalize.inria.fr`.

[18] M. Harrison, W. Ruzzo, and J. Ullman. Protection in operating systems. *Communications of the ACM*, 19:461–471, 1976.

[19] M. Jaume. Security rules versus security properties. In *Information Systems Security - 6th Int. Conf., ICISS*, volume 6503 of *LNCS*, pages 231–245. Springer, 2010.

[20] M. Jaume, V. Viet Triem Tong, and L. Mé. Flow based interpretation of access control: Detection of illegal information flows. In *Information Systems Security - 7th Int. Conf., ICISS*, volume 7093 of *LNCS*, pages 72–86. Springer, 2011.

[21] L. Lamport. How to write a proof. *AMM: The American Mathematical Monthly*, 102(7):600–608, 1995.

[22] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system, Documentation and user's manual*, release 3.07 edition, 2003.

[23] V. Prevosto and D. Doligez. Algorithms and proof inheritance in the Foc language. *Journal of Automated Reasoning*, 29(3-4):337–363, 2002.

[24] V. Prevosto and M. Jaume. Making proofs in a hierarchy of mathematical structures. In *11th Symp. on the Integration of Symbolic Computation and Mechanized Reasoning, Calculemus 2003*, pages 89–100. Aracne, 2003.

[25] R. Rioboo. Invariants for the Focal language. *Annals of Mathematics and Artificial Intelligence*, 56(3-4):273–296, 2009.

[26] N. Stouls and M. Potet. Security policy enforcement through refinement process. In *B 2007: Formal Specification and Development in B, 7th Int. Conf. of B Users*, volume 4355 of *LNCS*, pages 216–231. Springer, 2007.